PATENT DOCKET NO. GABBER 17-1-10-1

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

INVENTORS: **Eran Gabber, Lan Huang, Christopher Alexander Stein, and Elizabeth Shriver**

APPLICATION NO.: **09/848,826** CONFIRMATION NO. **7476**

FILED: **May 4, 2001** EXAMINER: **Ngoc V Dinh**

CASE NO.: **Gabber 17-1-10-1** GROUP ART UNIT: **2187**

TITLE: **IMPROVED FILE SYSTEM FOR CACHING WEB PROXIES**

CERTIFICATE OF MAILING/FACSIMILE

I hereby certify that this correspondence, along with any paper indicated as being enclosed, are deposited with the United States Postal Service as first-class mail, postage prepaid, in an envelope addressed to: Mail Stop Amendment, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

4 November 2004
Date

[illegible]
Diane A. Sears

Mail Stop Amendment
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

DECLARATION OF ERAN GABBER, LAN HUANG AND CHRISTOPHER ALEXANDER STEIN

We, Eran Gabber, Lan Huang, and Christopher Alexander Stein, do hereby declare the following:

1. We are three of the four co-inventors named in the above-identified patent application.

2. Before November 6, 2000, Eran Gabber, Christopher Alexander Stein and Elizabeth Shriver, the fourth co-inventor named on the above-identified patent application, submitted an invention disclosure to the appropriate personnel at Lucent Technologies, Inc. (Lucent), for consideration of obtaining a patent thereon. That submission eventually led to the filing of the above-identified patent application. A copy of that invention disclosure is attached hereto as

Exhibit "A." It comprises a Lucent "Disclosure of Invention" form as well as a paper entitled "Hummingbird: a light-weight file system for caching system web proxies" (hereinafter "the Hummingbird Paper") was prepared describing the invention.

3. Exhibit "A" describes the invention claimed in at least claims 1-10, 13-15, 20-22, 27, and 28 of the above-identified application. The date at the top of the Disclosure of Invention form has been redacted (as marked in Exhibit A), but is earlier than November 6, 2000. Another portion of the Disclosure of Invention form that does not disclose the invention also has been redacted (and also marked in Exhibit A).

4. As noted in section 5.3 of the Hummingbird Paper that forms part of Exhibit A, the invention disclosed in the Hummingbird Paper was reduced to practice prior to the date of the Hummingbird Paper.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of title 18 of the United States code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Date: 11/2/2004 — [illegible]
Eran Gabber

Date: ______ — ______
Lan Huang

Date: ______ — ______
Christopher Alexander Stein

M:\TNACCARELLA\CLIENTS\LUCENT\25748\PTO\DECLARATION 10-28-04 (DAS).DOC

E JC67 NOV 0 8 2004 PATENT & TRADEMARK OFFICE

PATENT **DOCKET NO. GABBER 17-1-10-1**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

INVENTORS: **Eran Gabber, Lan Huang, Christopher Alexander Stein, and Elizabeth Shriver**

APPLICATION NO.: **09/848,826** CONFIRMATION NO. **7476**

FILED: **May 4, 2001** EXAMINER: **Ngoc V Dinh**

CASE NO.: **Gabber 17-1-10-1** GROUP ART UNIT: **2187**

TITLE: **IMPROVED FILE SYSTEM FOR CACHING WEB PROXIES**

CERTIFICATE OF MAILING/FACSIMILE

I hereby certify that this correspondence, along with any paper indicated as being enclosed, are deposited with the United States Postal Service as first-class mail, postage prepaid, in an envelope addressed to: Mail Stop Amendment, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

4 November 2004
Date

[illegible]
Diane A. Sears

Mail Stop Amendment
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

DECLARATION OF ERAN GABBER, LAN HUANG AND CHRISTOPHER ALEXANDER STEIN

We, Eran Gabber, Lan Huang, and Christopher Alexander Stein, do hereby declare the following:

1. We are three of the four co-inventors named in the above-identified patent application.

2. Before November 6, 2000, Eran Gabber, Christopher Alexander Stein and Elizabeth Shriver, the fourth co-inventor named on the above-identified patent application, submitted an invention disclosure to the appropriate personnel at Lucent Technologies, Inc. (Lucent), for consideration of obtaining a patent thereon. That submission eventually led to the filing of the above-identified patent application. A copy of that invention disclosure is attached hereto as

Exhibit "A." It comprises a Lucent "Disclosure of Invention" form as well as a paper entitled "Hummingbird: a light-weight file system for caching system web proxies" (hereinafter "the Hummingbird Paper") was prepared describing the invention.

3. Exhibit "A" describes the invention claimed in at least claims 1-10, 13-15, 20-22, 27, and 28 of the above-identified application. The date at the top of the Disclosure of Invention form has been redacted (as marked in Exhibit A), but is earlier than November 6, 2000. Another portion of the Disclosure of Invention form that does not disclose the invention also has been redacted (and also marked in Exhibit A).

4. As noted in section 5.3 of the Hummingbird Paper that forms part of Exhibit A, the invention disclosed in the Hummingbird Paper was reduced to practice prior to the date of the Hummingbird Paper.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of title 18 of the United States code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Date:________________ ________________________
Eran Gabber

Date:________________ ________________________
Lan Huang

Date: Nov. 1, 2004 [illegible]
Christopher Alexander Stein

M:\TNACCARELLA\CLIENTS\LUCENT\25748\PTO\DECLARATION 10-28-04 (DAS).DOC

JC67 NOV 0 8 2004 U.S. PATENT & TRADEMARK OFFICE

PATENT **DOCKET NO. GABBER 17-1-10-1**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

INVENTORS: **Eran Gabber, Lan Huang, Christopher Alexander Stein, and Elizabeth Shriver**

APPLICATION NO.: **09/848,826** CONFIRMATION NO. **7476**

FILED: **May 4, 2001** EXAMINER: **Ngoc V Dinh**

CASE NO.: **Gabber 17-1-10-1** GROUP ART UNIT: **2187**

TITLE: **IMPROVED FILE SYSTEM FOR CACHING WEB PROXIES**

CERTIFICATE OF MAILING/FACSIMILE

I hereby certify that this correspondence, along with any paper indicated as being enclosed, are deposited with the United States Postal Service as first-class mail, postage prepaid, in an envelope addressed to: Mail Stop Amendment, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

4 November 2004 [illegible]
Date Diane A. Sears

Mail Stop Amendment
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

DECLARATION OF ERAN GABBER, LAN HUANG AND CHRISTOPHER ALEXANDER STEIN

We, Eran Gabber, Lan Huang, and Christopher Alexander Stein, do hereby declare the following:

1. We are three of the four co-inventors named in the above-identified patent application.

2. Before November 6, 2000, Eran Gabber, Christopher Alexander Stein and Elizabeth Shriver, the fourth co-inventor named on the above-identified patent application, submitted an invention disclosure to the appropriate personnel at Lucent Technologies, Inc. (Lucent), for consideration of obtaining a patent thereon. That submission eventually led to the filing of the above-identified patent application. A copy of that invention disclosure is attached hereto as

Exhibit "A." It comprises a Lucent "Disclosure of Invention" form as well as a paper entitled "Hummingbird: a light-weight file system for caching system web proxies" (hereinafter "the Hummingbird Paper") was prepared describing the invention.

3. Exhibit "A" describes the invention claimed in at least claims 1-10, 13-15, 20-22, 27, and 28 of the above-identified application. The date at the top of the Disclosure of Invention form has been redacted (as marked in Exhibit A), but is earlier than November 6, 2000. Another portion of the Disclosure of Invention form that does not disclose the invention also has been redacted (and also marked in Exhibit A).

4. As noted in section 5.3 of the Hummingbird Paper that forms part of Exhibit A, the invention disclosed in the Hummingbird Paper was reduced to practice prior to the date of the Hummingbird Paper.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of title 18 of the United States code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Date:________________ ________________________
Eran Gabber

Date: 11/1/2004 [illegible]
Lan Huang

Date:________________ ________________________
Christopher Alexander Stein

M:\TNACCARELLA\CLIENTS\LUCENT\25748\PTO\DECLARATION 10-28-04 (DAS).DOC

Class number: 3	12/20/99 Date

or IP-Law Use Only

[illegible]

We would like Sharon Fenick from Schnader Harrison Segal Lewis (Phil.) if possible.

LUCENT TECHNOLOGIES INC.

DISCLOSURE OF INVENTION

HIS DISCLOSURE SHOULD BE SUPPLEMENTED BY ATTACHING COPIES OF RELEVANT DOCUMENTS SUCH AS ECHNICAL MEMORANDA, PUBLISHED OR TO-BE-PUBLISHED ARTICLES AND ENGINEERING NOTEBOOK PAGES. (Also, if for any item there is insufficient space on the form, attach additional pages an necessary.)

DESCRIPTIVE TITLE OF INVENTION: A light-weight file system for caching web proxies

INVENTOR #1: Elizabeth Shriver
Name (Print) Company/Location
Phone/E-mail Director's Name

INVENTOR #2: Eran Gabber
Name (Print) Company/Location
Phone/E-mail Director's Name

NVENTOR #3: Christopher Stein
Name (Print) Company/Location
Phone/E-mail Director's Name

PRIMARY CONTACT
If more than one inventor is named above, who will have the primary responsibility for interfacing with Lucent IP-Law with respect to preparing and prosecuting a patent application for the invention?

Inventor Name: Elizabeth Shriver

FOR FILE
RECEIVED
OCT 13 2004
SYNNESTVEDT & LECHNER
ATTEN: TXN

. PRESENT STATE OF THE INVENTION

☐ Idea ☐ Research ☐ Development

☐ Manufacture (Product Name ______ Ship Date ______)

GOVERNMENT CONTRACT INVENTION
Was the invention made under a government contract? ☐ Yes [✓ No]

4. PRESENT STATE OF THE ART

Briefly describe the closest already-known technology that relates to the invention. This would include, for example, already existing products, methods or compositions which are known to you personally or through descriptions in publications or patents.

~~clustering~~ See related work section of paper

5. ADVANCEMENT IN STATE OF THE ART

Briefly describe the unique advancement achieved by the invention. This may be done, for example, by describing a problem with the prior art that is solved or specific objects that are achieved by the invention.

We developed a high-performance file system for caching web proxies; the problem we solve is how to get performance when the application is really a cache.

6. HOW ACHIEVED

Briefly describe the invention and how it achieves the advancement described in paragraph 6.

We cluster files together, amortizing disk access time.
We had a flat name space.

7. DISCLOSURE OUTSIDE OF LUCENT

Anticipated Publications Date: June 99 Publication Name: USENIX 2000

Submitted to Publication Clearance? [✓ Yes] No

If the invention was or will otherwise be disclosed to any non-Lucent employee, describe to whom (person/company), when, where, why, and whether it was/will be under a non-disclosure agreement.

8. Have any of the submitters discussed this invention with an attorney (either Lucent or outside) other than Jeffrey M. Weinick? Yes [✓ No]
If Yes, give name of the attorney:

9. Is this invention related to another Lucent invention submission, patent application, or issued patent? Yes [✓ No] (Please check Yes or No only. If Yes, attorney will contact you for further information.)

10. INVENTOR #1 [illegible]

Signature Date

INVENTOR #2 [illegible]

Signature Date

INVENTOR #3 [illegible] ~~Sept~~ Oct 12, 2004

Signature Date

Lucent Technologies
Bell Labs Innovations

INTELLECTUAL PROPERTY – LAW

Subject: Submission No. 121467

date: January 13, 2000

Title: Hummingbird: A Light-Weight File System For Caching Web Proxies

from: Jeffrey M. Weinick
Corporate Counsel
Intellectual Property - Law
MH 3B-507
(908) 582-2188
jweinick@lucent.com

E. Gabber:
E. Shriver:
C. Stein:

The above-identified patent submission, of which you appear to be an originator, was submitted on December 13, 1999. to consider its patentability.

Lucent's initial patent filing policy was to protect every invention submitted, through the filing of a patent application. The corporation's concern with cost control, however, has now caused us to focus somewhat more narrowly on those inventions that would seem to have the most patent licensing potential. Experience has taught us that even some highly sophisticated inventions are difficult to license and/or to secure significant royalty revenue from if they could be licensed. In this particular case, we have concluded after a careful review that the potential licensing value of a patent covering the subject matter in question is not sufficient to warrant the expense of obtaining patent protection.

We look forward to future submissions from you, which, as always, we will carefully consider. In the meantime, if I can be of any further assistance, please do not hesitate to contact me.

Jeffrey M. Weinick
[illegible]

Copy to:

A. Silberschatz

Lucent Technologies
Bell Labs Innovations

Bell Laboratories

Subject: Hummingbird: A light-weight file system for caching web proxies   date: 12/13/99

from: M. Reiter
Org. BL0112350
MH 2A-342
908-582-4328

J. Weinick:

Jeff:

Please consider the patentability of the attached "Hummingbird: A light-weight file system for caching web proxies". The following is a concise description of the patentable part of the paper.

The file system ideas for a non-persistant file system with access to mostly small files.

[illegible]

MH-10009634-MR -em   Mike Reiter

Attachments
Invention Disclosure Form
Paper

Hummingbird: A light-weight file system for caching web proxies

Eran Gabber
eran@bell-labs.com

Elizabeth Shriver
shriver@bell-labs.com

Christopher Stein
stein@eecs.harvard.edu

Abstract

Today, caching web proxies use general purpose file systems to store information that does not fit into main memory. Proxies such as Squid or Apache, when running on a UNIX system, typically use some derivative of the 4.2BSD Fast File System (FFS) for this purpose. FFS was designed 15 years ago for workload demands and requirements very different from that of a caching web proxy. Some of the differences are high temporal locality, relaxed persistence and a different read/write ratio. In this paper, we characterize the web proxy workload, describe the design of Hummingbird, a light-weight file system for web proxies, and present preliminary trace-driven performance measurements of Hummingbird. Our results indicate that Hummingbird's throughput is 7-8 times larger than a simulated version of Squid running on XFS, a high performance UNIX file system.

1 Introduction

Caching web proxies are computer systems dedicated to caching and delivering web content. Typically, they exist on a corporate firewall or at the point where an Internet Service Provider (ISP) peers with its network access provider. From a web performance and scalability point of view these systems have three purposes: reduce request load on origin servers, improve web client latency, and drive down the ISP's network access costs.

Proxy hit rates are slow to converge [Breslau99]. That is, the addition of more memory has less and less of an impact on the hit rate. The amount of memory required to reach hit rate convergence is economically infeasible, given current costs of main memory. So proxies use disks to store even larger amounts of data.

Squid and Apache are two popular web proxies. Both of these systems use the standard file system services provided by the host operating system. On UNIX this is most often a descendant of the 4.2BSD UNIX Fast File System (FFS) [McKusick84]. FFS was designed for workstation workloads and is not optimized for the different workload and requirements of a web proxy. Studies have observed that file system latency is a key component in the latency observed by web clients [Roussekov99].

Some commercial vendors have improved I/O performance by rebuilding the entire system stack; a special operating system with an application-specific file system executing on special hardware [CacheFlow99, NetApp99]. Needless to say, these solutions are expensive. We believe that a lightweight and portable library can be built that will allow proxies to achieve performance close to that of a specialized system on commodity hardware, within a general-purpose operating system, and with minimal changes to their source code.

We have built a simple, lightweight file system library named Hummingbird that runs on top of a raw disk partition. This system is easily portable—we have run it with minimal changes on IRIX, Solaris, and Linux. In this paper we describe the design, interface, and implementation of this system along with some experimental results that compare the performance of our system with a UNIX file system. Our results indicate that Hummingbird's throughput is 7–8 times larger than a simulated version of Squid running on XFS, a high performance UNIX file system (see Section 5).

Throughput the rest of this paper, we use the terms *proxy* or *web proxy* to mean *caching web proxy*. Section 2 presents some background on file systems and proxies. The information on the traditional bottlenecks of FFS is important because it motivates much of our design. Section 3 presents the important characteristics of the proxy workload that we considered for our file system. Section 4 describes the Hummingbird file system. Our experiments and results are presented in Section 5. Section 6 discusses related work in file systems and web proxy caching.

2 Background

The performance bottlenecks of traditional FFS are the disk head positioning times between blocks, small I/Os, and synchronous meta-data operations.

Disk head positioning is caused because files are not written in contiguous extents. Rather, files are a collection of fixed size blocks, typically 8 KB in size. A file meta-data structure known as the inode contains a series of pointers to these blocks. This is necessary to support file append and truncation and random access within the file. Seeks are caused between files when the reference stream locality does not correspond with the on-disk spatial locality. FFS attempts to reduce these seeks by locating the file blocks and meta-data close together on disk. This is the motivation behind *cylinder groups* and *directories*. Here, the responsibility for performance lies with the application or user who must construct a hierarchy with directory locality that matches future usage patterns. This is a challenge for web proxies because there is no human to provide information on reference locality. As well, in order to keep file lookup times down, the directory hierarchy must be well-balanced and a single directory should not have too many entries.

Squid attempts to balance the directory hierarchy, but in the process distributes the reference stream across directories, destroying locality. Apache uses URL string information to map files from the same origin server into the same directory. For specifying locality by a web proxy, a much more direct and low-overhead mechanism can be used.

FFS provides durability guarantees to applications. For example, under traditional UNIX semantics, a file create is durable once the create call returns. To achieve this the file system writes synchronously to disk. In addition, because the buffer cache is an unordered array of blocks, synchronous writes are necessary to enforce the ordering of meta-data updates so that the file system is recoverable. Since a web cache does not rely on file durability for correctness, it is free to trade it for performance. However, due to the logarithmic convergence of the web hit rate [Breslau99] it is desirable that much of the cache is recoverable. Couple the weak durability requirements with the whole-file access pattern and it is feasible for files to be written to disk in very large extents.

The use of traditional file systems also forces two architectural issues on the cache. First, the standard file system interface copies from kernel VM into the applications' address space. Second, the file system caches file blocks in its own buffer cache, where it does not have access to application-specific information. Web proxies manage their own application-level VM caches to save on memory copies and use private information to facilitate more effective cache management. However, web documents cached at the application level are likely to also exist in the file system buffer cache, especially if recently accessed from disk. This is the multiple buffering problem described in [Pai99]. Multiple buffering reduces the effective size of the memory cache, driving down the hit rate and degrading client latency. Administrators of web caches tend to increase the size of the application-level cache to before the point where process paging begins. A single unified cache solves the multiple buffering and configuration problems. Memory copy costs can be alleviated by passing data by reference rather than copying. Both of these can be done using a statically linked library that accesses the raw disk partition.

3 Web proxy workload characteristics

3.1 Details of the studied log

We studied a week's worth of web proxy logs from a major, national ISP, collected from January 30 to February 5, 1999. This proxy ran Netscape Enterprise server proxy software and the logs were generated in Netscape-Extended2 format. Over this one-week period, the proxy logged approximately 13.1 million HTTP requests. Of these over 98% were GETs. Of this, 34% were considered non-cacheable by the Netscape proxy. For the purpose of our analysis we isolate the request stream to those that would affect the file system underlying the proxy. Thus we exclude the 34% of the GET requests which are considered non-cacheable by the proxy. For file size analysis and simulation driving, we need the file size every time a new file is seen. Conditional GETs (304s) do not contain the size of the original content. So we process the log and look back to past references for the file sizes of 304 requests. This preprocessing results in the removal of about 4% of the log records, nearly all during the first few days. We eliminate the first few days and are left with 4 days of processed logs containing 4.8 million requests for 14.3 GB of unique cacheable data and 27.6 GB total requested cacheable data. Assuming an infinite file system cache, the proxy hit rate is 0.67. Since the time stamps in the log have a resolution of one second, our workload

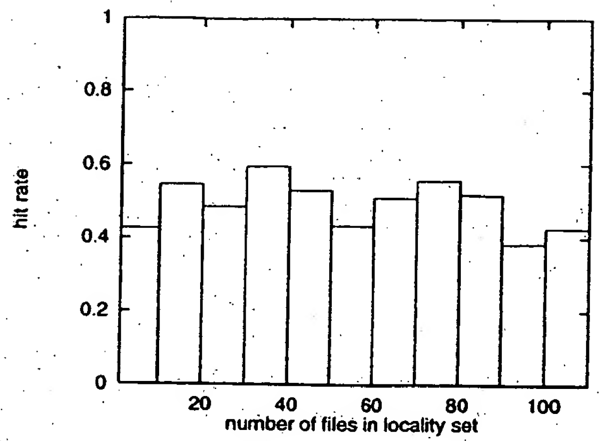

Figure 1: Hit rate vs. locality set size, where the size is expressed in number of files. The hit rate is the mean percentage of the locality set that is accessed.

generators also have a time granularity of one second.

3.2 Workload characteristics

Web proxy workloads have special characteristics, which are different than those of a traditional UNIX file system workload. This section describes the special characteristics of proxy workloads.

File access. Files are always sequentially accessed in their entirety.

File size. Most cacheable web documents are small. The median file size is approximately 2 KB and the average request size is 6167 B. Over 90% of the references are for documents smaller than 8 KB. This is in agreement with [Goldberg98], which shows that 80% of files are less than 10 KB and [Maltzahn99], which shows that 74% of files are less than 8 KB.

Spatial locality. Client web accesses are characterized by a request for an HTML page followed by requests for the embedded images. The set of images within a cacheable HTML page changes slowly over time. So if a page is requested by a client it makes sense for the server to anticipate future requests by prefetching the images historically associated with it.

We studied one day of the web proxy log for this reference locality. Requests are de-multiplexed on client IP address. On a particular client stream an HTML page and the following series of non-HTML files are grouped together into a *locality set*. These locality sets are the basis for capturing reference locality. For the analysis, we form static locality sets. The first time we see an HTML file, we coalesce it with all following non-HTMLs from the same client to form the primordial locality set, which does not change. The next time the HTML file is referenced we form another locality set in the same manner and compare its file members with those of the primordial locality set. Figure 1 shows the hit rate vs. locality set size. The hit rate across locality set sizes is stable. For large locality sets, faulting in entire locality set from secondary store, rather than file-by-file has a dramatic impact on average latency. However, most locality set references are small. 85% of the references are to locality sets with fewer than 20 files and 95% fewer than 40 files. The average hit rate across all references is 47%. Thus, on average, a locality set re-reference accesses almost half of the files of the original reference. This is an impressive number, because this is not a simple, global cache hit rate but, the hit rate for a series of requests from a very small and isolated set of files.

We also studied the size of the locality sets in bytes, and found that 42% are 32 KB or smaller, 62% are 64 KB or smaller, 73% are 96 KB or smaller, and 80% are 128 KB or smaller.

Table 1: Simulated idle time and disk activity on a single disk with data-only accesses.

simulated disk access time (ms)	simulated idle time per 1-minute interval (s)			disk activity per 1-minute interval (# operations)		
	min	max	median	min	max	median
12	34.5	59.8	49.7	20	2128	858
25	0.7	59.5	35.7	20	2120	753

Idleness. The web proxy workload tends to vary, as shown in Table 1. However, the request rate does not show the entire picture. There is ample idle time even at the busiest periods.

We used the following model to compute a lower bound on the amount of idle time experienced by a single disk that contains a file system that is accessed for every request in the processed log. We assumed that the disk contains only the data and no meta-data. Thus only the contents of the requests has to be read or written to/from the disk. No meta-data, such as directories, has to be updated. We further assumed that the disk has 64 KB blocks, and that a request of n bytes requires $\lceil \frac{n}{64KB} \rceil$ disk operations. The above assumptions correspond with the worst-case scenario in accessing the Hummingbird file system.

We counted the number of disk operations in each one-second interval in the processed log, and if this number exceeds the disk capacity in this interval, the remaining operations were carried to the next one-second interval. Table 1 shows the distribution of the idle time in one-minute intervals, where the idle time is the remaining time in the interval after performing the disk operations in that interval using the above algorithm. We used two disk access times: 12 ms per access, which corresponds to a Seagate ST118202FC disk (see Table 2), and 25 ms per access, which corresponds to twice the activity rate of our original load. In both cases, there was no one-minute interval in the entire log in which the disk was not idle. This is an important observation, since Hummingbird assumes that certain bookkeeping activities should be performed by background daemons during idle time (see Section 4.5). The disk activity is different for different simulated disk access times since some operations started in the last second of the one-minute interval and carried to the next interval.

Note that if the activity rate exceeds the capacity of a single disk, we can always add more disks and split the workload among them. In particular, Squid supports multiple disk caches. Another point to note is that a UNIX file system generates many than one disk operation for most requests, since it stores also the meta-data on the disk, and meta-data update operations are synchronous, which further slows down the processing.

4 File system design

The design of Hummingbird is influenced by proxy workload characteristics discussed in Section 3. Hummingbird uses locality hints generated by the proxy to pack files into large, fixed-size extents called *clusters*. These are the unit of disk access. Therefore, reads and writes are large, amortizing disk positioning times and interrupt processing.

Hummingbird manages a large memory cache. Since the file system is linked in with the proxy, no memory copies or memory mappings are required to move data from the file system to the proxy. The file system simply passes a pointer. Likewise, the proxy passes the file system a pointer when it writes data. This interface does not provide protection, but it is fast and flexible. We have only designed the file system for a single client, so protection is not necessary. The proxy may be multi-threaded, in which case access is serialized with a single lock on the file system meta-data. Using a single lock may slow the system under a heavy load, but we believe that it is adequate, since most file reads and writes return very quickly.

Hummingbird needs to clean the main memory and disk so that there is room for new files and clusters to be added. Since our workload characterization study shows that there is idle time, Hummingbird uses this idle time for daemons that clean main memory and disk.

While there are invariants across proxy workloads, some characteristics will change. We have designed Hummingbird to be configurable and flexible so that proxies in different environments can optimize the system to their workload. To this effect, the system has several "knobs" that the proxy is free to set to optimize the system for its workload. Some of these are set at file system initialization and others can be set during operation.

The parameters set at file system initialization include: size of a cluster, cluster threshold (the maximal internal fragmentation), memory cache threshold to begin cleaning, memory cache eviction policy, file hash table size, file and cluster lifetimes, disk data layout policy, and size and name of the disks. During run time, the proxy is free to set the lifetime of clusters or files. These values are used by the background daemons to decide whether to remove a file or cluster from the cache.

The file system functions, objects, and meta-data are discussed in the following section, along with the background processing.

4.1 Basic interface

The basic interface follows.

- **`int write_file(char* fname, void* buf, size_t sz);`**

  This function asynchronously writes the contents of the memory area starting at `buf` with size `sz` to disk with filename `fname`. It returns the size of the file. This file might be stored in a main memory cache. Once the pointer `buf` is handed over to the file system, the application **should not** use it again; the file system becomes responsible for the pointer and the block of memory it references.

- **`int read_file(char* fname, void** buf);`**

  This function sets `*buf` to the beginning of the memory area containing the contents of the file `fname`. It increments a reference count and returns the size of the file. If the file is not in main memory, another file or files may be evicted to make room, and the specified file will be read from disk. In this case, `read_file()` allocates the memory space to store the file. If the file is already in memory, then `read_file()` will return without any disk operation. The pointer to the file in main memory remains valid until the corresponding call to `done_read_file()` is made.

- **`int done_read_file(char* fname, void* buf);`**

  This function releases the space occupied by the file in main memory by decrementing the reference count. The space is uniquely identified by the filename and address of the memory area that was set when the `read_file()` was done. Every `read_file()` must be accompanied by a `done_read_file()`. Otherwise, the file will stay in memory indefinitely (until the application program terminates).

- **`int delete_file(char* fname);`**

  This function deletes the file named `fname` from the file system. The file must not have any active `read_file()`'s.

- **`int collocate_files(char* fnameA, char* fnameB);`**

  This function attempts to collocate file `fnameB` with file `fnameA` on disk. Both files must be previously written (by calling `write_file()`). This function may fail to collocate the files on disk due to reasons such as lack of contiguous space or the `collocate_files()` call arriving too late after `write_file()` calls.

All routines return a positive integer or zero if the operation succeeds; a negative return value indicate an error code, which encodes the type of error.

4.2 Extended interface

Large files are special. They account for a very small fraction of the requests, but a significant fraction of the bytes transferred. In the log we analyzed, files over 1 MB accounted for over 8% of the bytes transferred, but only 0.02% of the requests. Caching these large files is not important for the average latency perceived by clients, but is an important factor in the network access costs of the ISP. These files should be cached not to improve client latency, but to lower access costs. With this as the goal, it is better to store these large files on disk, and not in the file system cache in memory. Large files have a smaller average re-reference rate and a single large file, due to the heavy tail of the size distribution, will squeeze out many average sized files, many of which will later be re-faulted into memory.

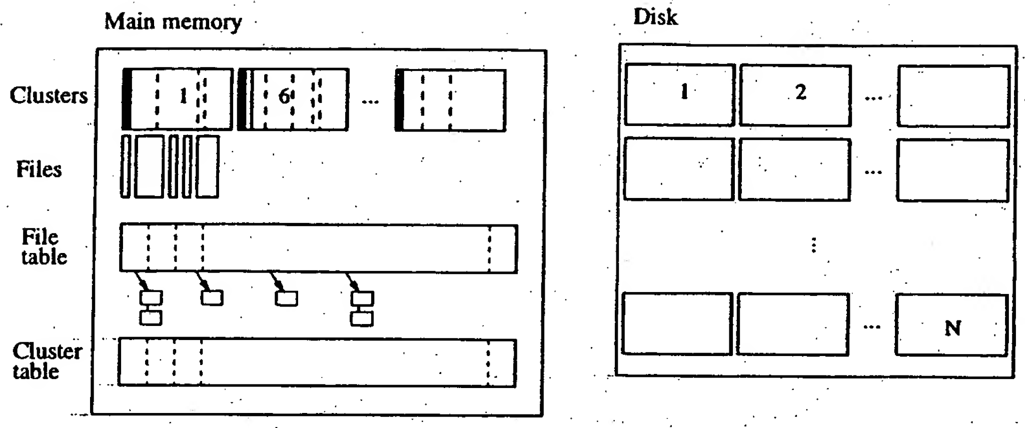

Figure 2: Contents of main memory (clusters, files, and meta-data) and disk (clusters) for Hummingbird.

- `int write_nomem_file(char* fname, void* buf, size_t sz);`

  This function bypasses the main memory cache and writes a file directly to disk. This file is flagged so that when it is faulted in, it does not compete with other documents for cache space and is immediately released after the proxy issues the `done_read_file()`.

Our disk-cleaning daemons delete files and clusters using the *file lifetime* and the *cluster lifetime*. The initial values set for these parameters might not be adequate for the workload; they might have to be made smaller to clean more disk space, or larger to increase the proxy hit rate.

- `int update_file_lifetime(int seconds)` and `int update_cluster_lifetime(int seconds);`

  These functions support modification of the lifetimes.

4.3 Memory objects

Hummingbird stores two types of main objects in main memory: *files* and *clusters*. A file is created by the file system when a `write_file()` is received. Clusters contain files and some file meta-data. The size of a cluster in tunable. Grouping files into clusters allows the file system to physically collocate files together since when a cluster is read from disk, all of the files in the cluster are read. A cluster is written to disk as soon as it is created. This way, clusters are *clean*, i.e., they can be evicted from main memory by reclaiming their space without writing to disk.

To support locality among files, the application sends `collocate_files(fnameA, fnameB)` calls. The file system saves the mapping until the files needs to be written to disk. The assignment of files to clusters occurs as late as possible; files are grouped together into clusters when space is needed in main memory. At this point, the file system attempts to write `fnameA` and `fnameB` in the same cluster.

When the file system is building a cluster, it determines which files to add using an LRU ordering according to the last time the file had been finished being read (i.e., when the `done_read_file()` call was received). If the least-recently-used file has a list of collocated files, then these files are added to the cluster if they are in main memory. If a file is on the collocation list, and already has been added to a cluster, it can still be added to the current cluster if the file is in memory. Thus, it is possible for a file to be a member of multiple clusters, and stored in multiple locations on disk.

4.4 Meta-data

Hummingbird requires three types of meta-data: file system meta-data, file meta-data, and cluster meta-data. Figure 2 shows the meta-data and data of Hummingbird.

File system meta-data. To determine when main memory space needs to be freed, we have counts on the amount of space used for storing the files and the file system data. To assist with determining which files and clusters to evict from main memory, Hummingbird maintains two LRU lists, one for files which have not yet been packed into clusters and another for clusters which are in memory.

File meta-data. A hash table stores pointers to the file information. For all files, we must save a file number (discussed below), status, and a reference count of the number of users that are currently reading the file. The file status field identifies whether the file is not in a cluster, in one cluster, or in multiple clusters. URLs are used for file names. URLs can be long, and since we have many small files the URLs would take up a large share of main memory if they were kept permanently resident. Thus, we save the file name alongside the file data in its cluster and not permanently in memory. This way, the URL will be availalbe only on disk when the cluster is cleaned from memory. Since the application interface uses filenames, we transform the filename into a 32-bit number using a hash function used to locate the correct file meta-data. Hash collisions can be detected by comparing the requested URL with the URL stored on disk, once the cluster is faulted in.

Until a file is a member of a cluster, the file name and file size need to be maintained as part of the file meta-data. Also needed, is a list of files that should be collocated with this file. If the file system policies contain processing at the file-level, e.g., deleting files based on the last access time, then additional meta-data is needed. Once a file is added to a cluster, the file meta-data must include the cluster ID and the reference count for that file.

Cluster meta-data. A cluster table contains information about each cluster on disk: the status, last-time accessed, and a linked list of the files in the cluster. The cluster status field identifies whether the cluster is empty, on disk, or in memory. For our file system, the cluster ID identifies the location of the cluster on disk. While a cluster is in memory, the address of the cluster in memory is needed. The last-time accessed is needed to determine the amount of time since the cluster was last touched.

Since most of the time, the majority of the clusters will be on disk or in memory, we group together clusters into contiguous groups of size 100, and save the number of free clusters in an array indexed by the group. This data structure decreases the amount of time needed to find a free cluster.

4.5 Daemons

In order to have main memory and disk space available when needed, Hummingbird has a number of daemons that make space available by packing files together into clusters, evicting data from main memory, and freeing space on disk. These daemons run when the file system is idle in a round-robin fashion, and can be synchronously called by a function if needed. These daemons are threads, not heavyweight processes, and execute in Hummingbird's address space.

Pack files daemon. If the amount of main memory used to stored files is at or over a tunable threshold, the `pack_files_daemon()` uses the file LRU list to create a cluster of files and write the cluster to disk. The daemon packs the files using the information from the `collocate_file()` calls, attempting to pack files from the same locality set in the same cluster. Clusters do not have to be packed so that they are completely full; a tunable threshold such as 75% is specified.

This daemon uses one of several *disk layout policies* to decide which cluster to pack. These include:

- *closest cluster.* This policy picks the closest free cluster to the previous cluster on disk accessed. This is an approximation of the free cluster with the shortest seek time.

- *closest cluster to previous write.* This policy picks the closest free cluster to the previous cluster written to on disk. This will group together clusters that are written at similar times.

- *log-structured like.* This policy will overwrite the next cluster.

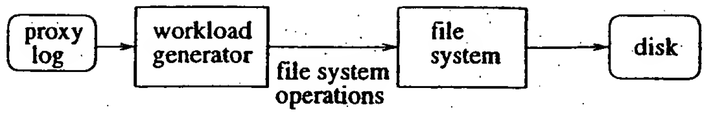

Figure 3: Simulation environment.

Free main memory data daemon. It is undesirable that before every use of main memory, the file system must evict some data out of memory. Instead, we have the `free_main_memory_data_daemon()` which evicts data when the amount of main memory used by the file system is at or over a tunable threshold. This daemon determines what to evict using a last-time accessed time stamp on the files and clusters in the LRU lists. If there are files that are older than the oldest cluster, the daemon will pack files together in a cluster as the `pack_files_daemon()` does. If a cluster is evicted, since it is clean, it need not be written to disk.

Free disk space daemon. There are a number of ways that we can free disk space. For example, the application can specify a *file lifetime* (set in the *delete_file_policy*), where a file should be automatically deleted once it exceeds the lifetime. Deleting could also occur on a cluster basis using the *cluster lifetime*, set in the *delete_cluster_policy*, i.e., once all files in the cluster have exceeded the lifetime, the cluster is deleted.

Clearly, if the file lifetime is set to a very large value, all files could be younger than the lifetime, and no files deleted when disk space is needed. If this occurs, a `read_file()` that attempts to evict files to disk will return an error code.

This daemon is not called if the disk layout policy is log-structured writing.

4.6 File system recovery

Web proxies are slowly convergent. It takes days to populate and stabilize a cache so that it operates at its maximal hit rate. So it is important that the cache contents are not completely lost after a failure. At the same time recovery must be quick. With today's disk sizes, the system cannot wait for a full disk scan before returning to normal operation.

Hummingbird does not have a directory structure and there are no meta-data consistency dependencies across clusters. There is no need for an analog of the UNIX `fsck` utility to ensure the consistency of the file system after a crash. The file system will be consistent. However, since all in-memory meta-data is lost during the crash, the system will not know what is stored on disk.

There are two solutions to this problem. The first is to maintain an auxiliary write-ahead log [Gray93], recording changes to the disk cache. After a crash, the meta-data is reconstructed by standard database log recovery techniques. This solution allows for fast recovery and reconvergence of hit rate at the cost of the run time expense of logging and additional system complexity. The second solution is to scan the disk in the background after the system recovers. This solution is simpler and does not affect performance during the normal operation of the system. However, if the system is very busy after a crash it may not be possible to find idle time to scan the disk and update the meta-data structures. Even if such idle time exists, the hit rate will be more slowly convergent. We have not firmly committed Hummingbird to one of these solutions.

5 Experimental results

We have built a simulation environment to run experiments comparing the performance of FFS and Hummingbird, and to compare Hummingbird policies. This environment consists of four components, as depicted in Figure 3. (1) The processed *proxy log* which was discussed in Section 3.2. (2) The *workload generator* simulates the operation of a caching web proxy. It reads the proxy log events and generates the file system operations that a proxy would have generated during processing of the original HTTP requests. (3) The *file system*, which is either Hummingbird or FFS. (4) The *disk*, which is a physical magnetic disk that the file system uses to store the files.

We considered using Polygraph instead of a real-world proxy log, but found that the PolyMix-1 traffic model does not model spatial locality of URLs in terms of HTML pages and their embedded images [Rousskov99b]. The workload generators, implementation details, and experiments are described in the following section.

5.1 Workload generators

We developed two workload generators: wg-Hummingbird which issues Hummingbird calls, and wg-Squid which mimics Squid's interaction with the file system. The generators take as input the modified proxy access log. The workload generators operate in a log-driven loop which processes logged events sequentially without pause[1]. This simulates a heavily loaded server.

If our logs contained time-to-live or expire information, our workload generators could delete files as appropriate.

Hummingbird workload generator: wg-Hummingbird. Each iteration of the wg-Hummingbird loop parses the next sequential event from the log and attempts to read the URL from Hummingbird. If successful, it continues and explicitly releases the file pointer. If the read attempt fails, it attempts to write the URL into the cache; this would have occurred after the proxy would have fetched the URL from the server.

The wg-Hummingbird maintains a hash table keyed by client IP address to store information for generating the collocate hints. The values in the hash table are the URLs of the most recent HTML file request seen from a particular client. The hash table only stores the URLs of static HTML files[2]. As requests for non-HTML documents are processed, wg-Hummingbird generates `collocate_files()` calls for the non-HTML paired with its client's current HTML file, as stored in the hash table. This is best illustrated by a few lines of pseudo-code, where `url` is the URL string of the event being processed and `client` is the IP address of the machine that originated the request. This logic takes place after the file system write of `url` has returned successfully.

```
if (url is a static HTML)
    hash_insert(url)
else
    if (html = hash_lookup(client))
        collocate_files(html, url)
```

Proxies must take care to keep the size of the hash table bounded. They can do this by maintaining an LRU threaded through the hash table records and removing records once the memory use reaches a threshold.

FFS workload generator: wg-Squid. The wg-Squid simulates the file system behavior of Squid-2.2. The Squid cache has a 3-level directory hierarchy for storing files; the number of children at each level is a configurable parameter. In order to minimize file lookup times, Squid attempts to keep directories small by distributing files evenly across the directory hierarchy.

When a file is written to the cache a top-level directory, or *SwapDir*, is selected. Squid attempts to load balance across the SwapDirs. Once the SwapDir has been selected, the file is assigned a file number which uniquely identifies the file within the SwapDir. The value of this file number is used to compute the names of the level-2 and level-3 directories. Thus, Squid does not use the URL or URL reference locality for file placement into directories, limiting the ability of the file system to collocate files which will be accessed together. Once the directory path has been determined, the file is created and written.

Squid has configurable high and low water marks. When the total cache size passes the high water mark, eviction begins. Files are deleted from the cache in modified LRU order with a small bias towards expired files. Eviction continues until the low water mark is reached.

Our workload generator simulates all this behavior except for the modified LRU. We do not have expires information in our log so we use standard LRU.

We selected the parameters of the wg-Squid based on the recommended values given in the Squid users mailing list [Squid99]. We used the following parameters:

- One cache directory on a single physical disk.
- 256 level-2 directories per top-level directory.

[1] We call this timing mode THROTTLE. We have implemented two other different timings modes: REAL and FAST. REAL issues the event with the frequency that they were recorded. FAST processes the events with a speed-up heuristic parameterized by n; if the time between events is longer than time n, then we only wait time n between them.

[2] We considered URLs with the suffixes ".html", ".htm", ".map", ".shtml", ".jsp", and "/" to be static HTML pages.

Table 2: Basic disk performance numbers for the 18 GB disks.

disk type	mean seek time	mean rotational latency	mean transfer rate
ST118202FC	5.2 ms	2.99 ms	19 MB/s
ST318275LC	6.9 ms	4.16 ms	12.5 MB/s

- 8 top-level directories for each 1 GB of disk space, which is a close approximation to the recommended formula $1 + \frac{dir\_size \times 2}{avg\_obj\_size \times L2 \times L2}$, where *dir_size* is the cache directory size in bytes, *avg_obj_size* is the average cached object size (6167 in our case) and $L2$ is the number of level-2 directory entries (256 in our case).
- 50 MB of main memory cache, which is computed by the formula 2MB + 15 *seconds of traffic.* We measured the 99.9 percentile of one-second interval traffic to be around 3MB/second in the processed log.

5.2 Hummingbird Implementation

The current implementation of Hummingbird has several limitations, which we plan to address in the near future. In particular, Hummingbird currently supports only a single disk, does not support files that are larger than a cluster size, and it is not multi-threaded.

Since Hummingbird is called by a workload generator that is fed by a log, it detects idle time when all of the requests with the same time stamp (with a 1-second resolution) were completed in less than one second of actual elapsed time. Hummingbird uses this idle time to call the main daemon; this daemon function calls the 3 daemons in a round-robin fashion.

5.3 Experiments

Most of our experiments ran on an SGI Origin 2000 with 18 GB Seagate Cheetah disks (ST118202FC) under IRIX 6.5. Some of our experiments were run on a SUN Ultra 60 with a 18 GB Seagate Barracuda (ST318275LC), due to unavailability of the SGI. Mean performance values of the disks are in Table 2. Both of these disks support delayed writes, hence a disk write returns as soon as the data is in the disk cache.

For practical reasons related to system availability, we have chosen to compare Hummingbird with XFS. XFS is a high-performance journalling file system that runs in the IRIX operating system, SGI's version of UNIX [Holton94]. XFS interacts with the kernel through the traditional VFS and vnode interfaces [Kleiman86].

Our experiments used the 4-day web proxy log as input into the workload generators. Since Hummingbird only supports files of 64 KB and smaller, we throw away the larger files. Among other measurements, we measure the ***proxy hit rate***, which represents how frequently the web page will not have to be fetched from the server, and the ***file system read time***, which represents how long the user must wait to see a web page if the file system has the file.

Comparing Hummingbird settings and policies. We compared different cluster sizes, different data layout policies, and different values of the cluster lifetime to determine the optimal parameter setting for our web proxy trace. Unless stated otherwise, these experiments use 256 MB of main memory and 4 and 9 GBs of the disk; the data layout policy is closest; the cluster size is 64 KB; the cluster lifetime is 60 minutes. By running in THROTTLE mode, the cluster lifetime does not map to "real" values, i.e., setting a file lifetime to 60 minutes represents 60 minutes of experiment time, not real time. Most of the experiments were on the SGI platform. Since the total amount of unique data for our 4-day log file is 14.3 GB, disks smaller than 14.3 GB force data to be deleted.

We experimented with a number of different sizes for the cluster; Table 3 shows our results for these experiments. The three cluster sizes have similar proxy hit rates, both in terms of files (presented in Table 3) and bytes (39–41%). We see that the larger clusters do not substantially reduce the number of disk reads, in fact, the cluster size of 96 KB has more disk reads, perhaps due to having to evict clusters that would soon be referenced

Table 3: Experiment results when varying the cluster size: 9 GB disk, 256 MB of main memory, cluster lifetime of 60 minutes, closest data layout policy.

cluster size (KB)	proxy hit rate	main memory hit rate	read_file() time (ms)	write_file() time (ms)	# of disk reads	total disk read time (ms)	# of disk writes	total disk write time (ms)	simulation run time (s)
64	0.65	0.74	3.32	0.21	820449	10261431	195827	679558	11720
96	0.66	0.73	4.05	0.23	841001	12690560	140043	624392	14261
128	0.64	0.74	4.74	0.28	812987	14477966	113034	623528	16536

Table 4: Experiment results when varying the data layout policy: 64 KB cluster size, 256 MB of main memory, cluster lifetime of 30 minutes (4 GB disk) and 60 minutes (9 GB disk).

data layout policy	disk size	read_file() time (ms)	write_file() time (ms)	mean disk read time (ms)	mean disk write time (ms)	simulation run time (s)
closest	4 GB	2.82	0.20	11.65	3.67	9926
closest to previous write	4 GB	2.91	0.30	12.12	3.43	11457
closest	9 GB	3.32	0.21	12.51	3.47	11720
closest to previous write	9 GB	3.40	0.20	13.01	3.47	11983

to free main memory due to the larger cluster size. Moreover, larger cluster sizes increased the `read_file()` time. This is the result of longer transfer sizes and approximately the same number of disk reads.

The current Hummingbird implementation truncates locality sets at the cluster size, i.e., if a locality set is larger than the cluster size, only some of the files are accessed together on disk in a cluster. The disk read and write time trends seen in Table 3 are as expected; the total amount of time spent reading increases as the cluster size increases since more "undesired" data is read due to Hummingbird packing multiple locality sets into the same cluster for small locality sets. We consider the 64 KB cluster size to be the best since the mean `read_file()` and `write_file()` times and the approximate throughput (# of requests / simulation run time) are the smallest.

We experimented with the *closest* and *closest-to-previous-write* data placement policies defined in Section 4.5. When experimenting with 4 GB of the disk, we had to change the cluster lifetime to 30 to guarantee enough free disk clusters. The proxy and main memory hit rates were the same for both policies as expected. Also, as expected, the disk cleaning daemon is slower for the closest-to-previous-write policy (2.14 ms compared to 1.47 ms) since more clusters have to be checked due to the recently written clusters being grouped together on disk. Table 4 shows the results for these experiments. Since there are 4–7 times more disk reads that disk writes (as illustrated in columns 6 and 8 in Table 3), the closest-to-previous-write data placement policy causes additional disk seeks, as seen by the larger times for both the file operations (`read_file()` and `write_file()`) and disk operations. The results indicate that the closest data placement policy is superior to the closest-to-previous-write policy.

We compared different settings for the cluster lifetimes on our Solaris platform. As seen in Table 5, both the proxy and main memory hit rates vary greatly. As the cluster lifetime decreases, more clusters are deleted by the `free_disk_space_daemon`. So, files are deleted that belong to these clusters, and the percentage of main memory space that can be used for storing files and clusters increases as seen in the 4th column in Table 5. **Note to PC: the 4th column will be filled in for the final paper.** This causes the main memory hit rate to increase as the cluster lifetime decreases. The simulation run time increases as the cluster lifetime increases because it takes more time to clean the disk and find free clusters. The file system could not find a free cluster on disk after servicing 3.6 million requests with a cluster lifetime setting of 150.

Thus, for the trace that we used, the best choice of policies for Hummingbird with 256 MB main memory and a 9 GB disk are: 64 KB cluster size, cluster lifetime of 80 minutes, and the closest data layout policy. (The cluster lifetime of 80 minutes was determined using experiments on the IRIX platform similar to those presented in Table 5.)

Table 5: Experiment results when varying the cluster lifetime: 9 GB disk, closest data layout policy, 64 KB cluster size, 256 MB of main memory.

cluster lifetime (minutes)	proxy hit rate	main memory hit rate	% of main memory used for meta-data	# of read_file() calls	read_file() time (ms)	# of write_file() calls	write_file() time (ms)	simulation run time (s)
5	0.53	0.88		2561251	1.63	2248644	0.11	5299
30	0.60	0.80		2866611	2.73	1943284	0.11	8847
60	0.62	0.77		2992017	3.12	1817878	0.11	10315
120	0.65	0.75		3107584	3.36	1702311	0.11	11396
135	0.65	0.75		3124873	3.39	1685022	0.11	11543

Table 6: Comparing Hummingbird with XFS with 256 MB of main memory.

file system	disk size	proxy hit rate	FS read time (ms)	FS write time (ms)	# of disk I/Os	mean disk I/O time (ms)	simulation run time (s)
Hum	4 GB	0.62	2.82	0.20	922871	9.85	9926
XFS	4 GB	0.64	16.67	14.98	10348201	7.26	78243
Hum	9 GB	0.66	3.30	0.21	1028922	10.77	11861
XFS	9 GB	0.67	18.08	12.24	9652591	7.88	78984

Comparing Hummingbird with XFS. We compared Hummingbird with XFS for 3 different disk sizes. Table 6 presents results where wg-Squid and the IRIX buffer cache together use 256 MB of main memory[3], Hummingbird has 256 MB of main memory, and Hummingbird has the best choice of policies as previously discussed.

The user-perceived latency is the 4th column, the FS read time. Hummingbird's smaller FS read time is due to the hits in main memory caused by grouping files in locality sets into clusters. Hummingbird's smaller FS write time is due to asynchronous disk writes and a single disk write writing multiple files to disk, and due to XFS having synchronous log writes.

Since we cannot compute the main memory hit rate for XFS, we instead compute the mean disk I/O time. We used the sar system utility on IRIX to report the disk activity. Note that the number of disk I/Os in XFS is larger than the total number of request in the log. This is because file operations resulted in multiple disk I/Os. This also explains why XFS read and write operations are slower than disk I/Os.

We can approximate the throughput for a 4 GB disk for Hummingbird as 485 requests/second, and XFS as 61 requests/second. This is not quite a fair comparison since the proxy hit rate is lower with Hummingbird. (We do not expect the simulation run time to increase more than 10% when the Hummingbird policies are set so that we would have a 0.64 hit rate.) The approximate Hummingbird throughput for a 9 GB disk is 406 requests/second, and XFS is 61 requests/second. Using approximate throughput as a comparison metric, we see that Hummingbird is 6.7–7.9 times faster than our simulated Squid running on XFS.

6 Related work

Related work falls into two categories: first, analyses of traditional FFS-based systems and ways to beat their performance limitations, and second, analyses of the behavior of web proxies and how they can better use the underlying I/O and file systems.

The first set of research extends back to the original FFS work of [McKusick84] which addressed the limitations of the System V file system by introducing larger block sizes, fragments, and cylinder groups. With increasing

[3]wg-Squid has 50MB of memory cache and the IRIX buffer cache has about 206 MB of memory. We pinned the rest of the systems physical memory to to limit the size of IRIX buffer cache.

memory and buffer cache sizes, UNIX file systems became able to satisfy more reads out of memory. LFS [Rosenblum92] sought to improve write time by packing dirty file blocks together and writing to an on-disk log in large extents called *segments*. The LFS approach necessitates a cleaner daemon to coalesce live data and free on-disk segments. As well, new on-disk structures are required. The FFS clustering work of [McVoy91] sought to improve write times not by changing the on-disk data structures, but by lazily writing the data, then batching it to disk in contiguous extents called *clusters*. Recent work in soft updates [Ganger94] and journalling [Hagmann87, Chutani92] has sought to alleviate the performance limitations due to synchronous meta-data operations, such as file create or delete, which must modify file system structures in a specified order. Soft updates maintains dependency information in kernel memory to order disk updates. Journalling systems write meta-data updates to an auxiliary log using the write-ahead logging protocol. This differs from LFS, in which the log contains all data, including meta-data. LFS also addresses the meta-data update problem by ordering updates within segments.

[Roussekov99] studied the performance of Squid and its use of the file system. [Maltzahn97] compared the disk I/O of Apache and Squid and concluded that they were remarkably similar. A follow-up simulation paper [Maltzahn99] recommended changes to the way that Squid names files and the use of VM and memory mapping, rather than the file system, for small files. [Markatos99] presents methods for web proxies to work around costly file system file opens, closes, and deletes. One of their methods, LAZY-READS, gathers read requests n-at-a-time, and issues them all at the same time to the disk; results are presented when n is 10. This is similar to our clustering of locality sets, since a read for a cluster will, on average, access 8 files. We feel that Hummingbird in a more general solution to the decreasing the effect of costly file system operations on a web proxy.

[Pai99] developed a kernel I/O system called IO-lite to permit sharing of "buffer aggregates" between multiple applications and kernel subsystems. This system solves the multiple buffering problem, but applications must use the I/O-lite interface that supersedes the traditional UNIX read and write operations.

7. Conclusions and future work

This paper explored file system support for applications with limited persistence requirements. A distinguishing feature of this file system is that it need not recover all data after a crash. Such a file system is especially useful for local caching of data, where permanent storage of the data is available elsewhere. A caching web proxy is the prime example an application that may benefit from this file system.

We implemented Hummingbird, a light-weight file system that is designed to support caching web proxies. Hummingbird has two distinguishing features: it stores its meta-data completely in memory, and it stores groups of related objects (e.g., HTML page and its embedded images) together on the disk. Our results are very promising; Hummingbird's throughput is 7–8 times larger than a simulated version of Squid running on XFS, a high performance UNIX file system.

We would like to extend our experiments with Hummingbird and test other implementations of FFS, such as those in Solaris and BSD. We also would like to study the impact of duplicating common files in multiple clusters.

Code available. We plan to make two of our tools, `wg_Squid` and `process_ne2log.pl`, publicly available. `process_ne2log.pl` takes a log in Netscape-Extended2 format and modifies it so that it can be used as input to `wg_Squid`.

Acknowledgements. Many thanks to Arthur Goldberg for giving us copies of the ISP log files that we studied. WeeTeck Ng helped us with some of the file system implementation. Bruce Hillyer and Philip Bohannon were very helpful in initial design discussions.

References

[Breslau99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: evidence and implications. *Proceedings of IEEE INFOCOM'99* (New York, NY), pages 126–134, March 1999.

[CacheFlow99] 1999. `http://www.cacheflow.com/products/`.

[Chutani92] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode file system. *Proceedings of the Winter 1992 USENIX Conference* (San Francisco, CA), pages 43–60, Winter 1992.

[Ganger94] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, CA), pages 49–60, November 1994.

[Goldberg98] Arthur Goldberg, Ilya Pevzner, and Robert Buff. Characteristics of internet and intranet web proxy traces. *Proceedings of the 24th International Conference on Technology Management and Performance Evaluation of Enterprise-Wide Information Systems (CMG98)* (Anaheim, CA), December 1998.

[Gray93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[Hagmann87] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (Austin, TX), pages 155–162, November 1987. In *ACM Operating Systems Review 21*:5.

[Holton94] Mike Holton and Raj Das. *XFS: a next generation journalled 64-bit filesystem with guaranteed rate I/O.* Technical report. SGI Inc, 1994. `http://www.sgi.com/Technology/xfs-whitepaper.html`.

[Kleiman86] S. Kleiman. Vnodes: an architecture for multiple file systems in Sun UNIX. *Proceedings of the USENIX Conference*, Summer 1986.

[Maltzahn97] Carlos Maltzahn, Kathy Richardson, and Dirk Grunwald. Performance issues of enterprise level proxies. *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)* (Seattle, WA), pages 13–23, June 1997.

[Maltzahn99] Carlos Maltzahn, Kathy J. Richardson, and Dirk Grunwald. Reducing the disk I/O of web proxy server caches. *Proceedings of the 1999 USENIX Annual Technical Conference* (Monterey, CA), pages 225–238, June 1999.

[Markatos99] Evangelos P. Markatos, Manolis G.H. Katevenis, Dionisis Pnevmatikatos, and Michail Flouris. Secondary storage management for web proxies. *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems* (Boulder, CO), pages 93–114, October 1999.

[McKusick84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[McVoy91] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. *Proceedings of the Winter 1991 USENIX Conference* (Dallas, TX), pages 33–43, January 1991.

[NetApp99] 1999. `http://www.netapp.com/products/netcache/`.

[Pai99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (New Orleans, LA), pages 15–28, February 1999.

[Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[Roussekov99] Alex Rousskov and Valery Soloviev. A performance study of the Squid proxy on HTTP/1.0. *World-Wide Web Journal, Special Edition on WWW Characterization and Performance and Evaluation*, 2(1–2), 1999.

[Rousskov99b] Alex Rousskov, Duane Wessels, and Glenn Chisholm. The first IRCache web cache bake-off – the official report, April 1999. `http://bakeoff.ircache.net/bakeoff-01/`.

[Squid99] Henrik Nordstrom (Squid Hacker). Squid mail archive, November 1999. `http://squid.nlanr.net/-mail-archive/squid-users/199911/0517.html`.